LIS Developer's Guide

Submitted under Task Agreement GSFC-CT-2

Cooperative Agreement Notice (CAN) CAN-00OES-01

Increasing Interoperability and Performance of
Grand Challenge Applications in the Earth, Space, Life, and Microgravity Sciences

January 3, 2008

Version 3.0

History:

Revision	Summary of Changes	Date
5.0	LIS5.0 public release	Jan 1, 2008
4.1	Data assimilation capability	Mar 5, 2005
3.0	Milestone "G" submission	May 7, 2004
2.3	LIS 2.3 code release	December 19, 2003
	Initial revison	

National Aeronautics and Space Administration
Goddard Space Flight Center
Greenbelt, Maryland 20771

Contents

1 Introduction

This document describes some of the interoperable features in LIS and how to use/extend them. The following sections describe the general development and documentation practices recommended for using and extending LIS software, followed by the guidelines for using the extensible features in LIS for customization and improved functionality.

2 Background

This section provides some general information about the LIS project and land surface modeling.

2.1 LIS

Land surface models provide characterizations of the water and energy exchanges and biogeochemical processes of the soil-vegetation-snowpack medium. A realistic representation of these processes is critical for improving the understanding of the boundary layer and land-atmosphere interactions. The development of LIS has been motivated by the need to develop an infrastructure that combines the use of land surface simulation, available observations and the required computing tools for accurate land surface prediction. As discussed in [6], LIS integrates and extends the capabilities of Land Data Assimilation Systems (LDASs) such as the 25km Global Land Data Assimilation System (GLDAS) and the 12.5km North American Land Data Assimilation System (NLDAS). LIS is primarily an infrastructure for operating an ensemble of land surface models with capabilities for data integration and assimilation, over user-specified regional or global domains. The new phase in LIS development is to extend its capabilities by linking with other earth system components, enabling coupled systems that can model land-atmosphere interactions more effectively.

LIS is designed using advanced software engineering principles, and features a highly modular, flexible, object oriented, component-based framework. Figure 1 shows the software architecture of LIS. The core of the system consists of structures to manage generic utilities such as time, configuration, geospatial transformations, I/O, parallel computing constructs, logging, etc. These structures provide generic, model-independent support for high performance computing, resource management, data and I/O handling, and other functions. The LIS core controls the overall program execution and manages the inclusion of user-defined extensible components through several related abstractions. These abstractions, shown in the middle layer, include generic representations of land surface models, data assimilation schemes, meteorological forcing schemes, domains, running modes etc. The specific user defined components extend these abstractions. For example, Figure 1 shows a number of land surface models (Noah, CLM, HySSIB, Catchment) implemented in LIS through the land surface model abstraction. By providing a structure that allows the reuse and community sharing of modeling tools, LIS allows rapid prototyping and development of new applications. These interoperable features in LIS has enabled the incorporation of a growing suite of community LSMs, meteorological forcing analyses, different sources of land surface parameters, and data assimilation schemes. The system also allows for the plug and play of various user-defined components and has enabled several intercomparison studies involving land surface models, parameters, and assimilation schemes.

Please refer to [6, 5, 7] for details on the design of LIS. This document provides instructions on the use of the "plug-and-play" features or abstractions in LIS.

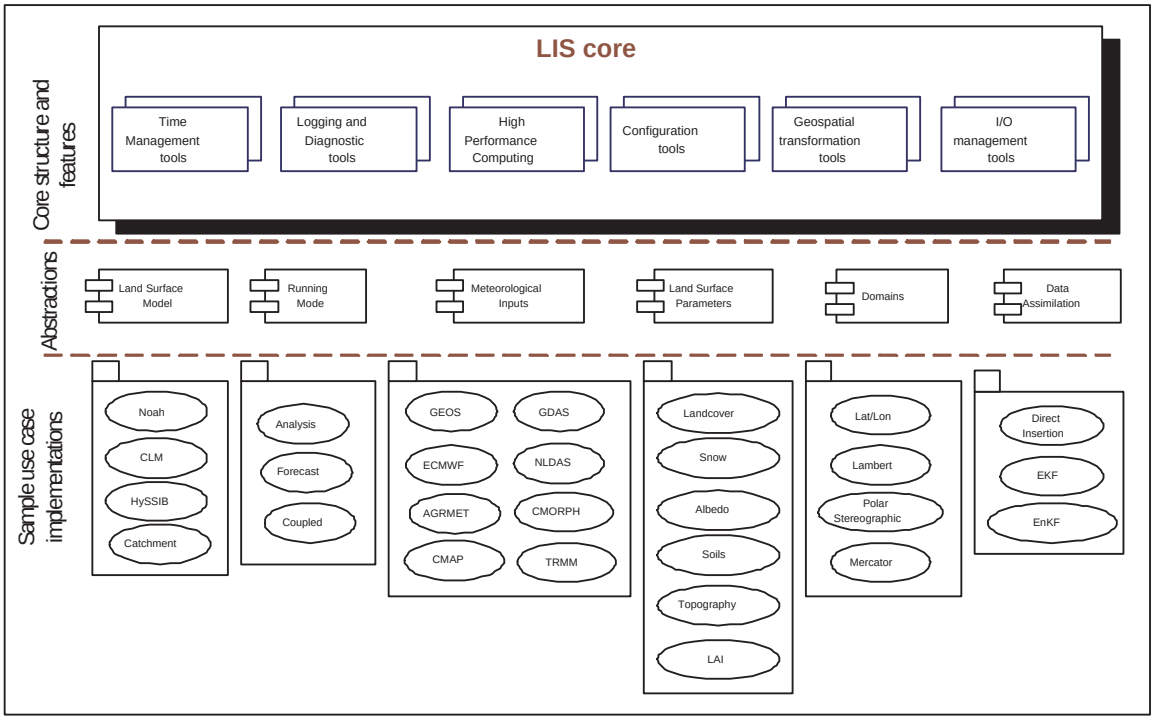

Figure 1: Software architecture of the LIS framework

3 Coding and Documentation Conventions

This section describes some of the coding and documentation conventions [1] that are helpful for developers of LIS.

3.1 Coding conventions

LIS is implemented using the Fortran 90 and C programming languages. Since different Fortran compilers parse source files differently depending on the file extension (such as .f, .f77, .F, .f90, and .F90) the task of porting code to different platforms is a difficult process. Therefore, Fortran additions and contributions to LIS code are expected to be written using the Fortran 90, and the sources files must have an F90 extension. Some of the style guidelines followed in LIS are as follows:

- Preprocessor: C preprocessor (cpp) is used wherever the use of a language preprocessor is required. The Fortran compiler is assumed to have the ability to run the preprocessor as part of the compilation process. The preprocessing tokens are written in uppercase to distinguish them from the Fortran code.
- Loops: All loops in Fortran are structured using do-enddo constructs as opposed to numbered loops.
- Indentation: Code with nested if blocks and do loops are expected to be indented for readability.
- Modules: Modules must be named the same as the file in which they reside. This is enforced due to the fact that make programs build dependencies based on file names.
- Implicit none: All variables in different modules should be explicitly typed, and this should be enforced by the use of the "implicit none" statement.

3.2 Documentation conventions

LIS uses an in-line documentation system that allows users to create both web-browsable (html) and print-friendly(ps/pdf) documentation. Each function, subroutine, or module includes a prologue instrumented for use with the ProTex auto-documentation script [2]. The following examples describe the documentation templates used in LIS.

Templates for routines that are not internal to modules.

```
!-------------------------------------------------------------------------
!         NASA/GSFC Land Information Systems LIS 5.0
!-------------------------------------------------------------------------
!BOP
!
! !ROUTINE:
!
! !INTERFACE:
!
! !USES:
!
! !INPUT PARAMETERS:
!
! !OUTPUT PARAMETERS:
!
! !DESCRIPTION:
!
! !BUGS:
!
! !SEE ALSO:
!
! !SYSTEM ROUTINES:
!
! !FILES USED:
!
! !REVISION HISTORY:
!
!  27Jun02  Username Initial specification
!
!EOP
!-------------------------------------------------------------------------
!BOC
!EOC
```

Template for a module :

```
!-------------------------------------------------------------------------
!         NASA/GSFC Land Information Systems LIS 5.0
!-------------------------------------------------------------------------
!BOP
!
! !MODULE:
!
! !PUBLIC TYPES:
!
! !PUBLIC MEMBER FUNCTIONS:
!
! !PUBLIC DATA MEMBERS:
!
! !DESCRIPTION:
!
! !REVISION HISTORY:
!
!  27Jun02  Username Initial specification
!
!EOP
```

Template for a C file:

```
//------------------------------------------------------------------------
//         NASA/GSFC Land Information Systems LIS 5.0
//------------------------------------------------------------------------
//BOP
//
// !ROUTINE:
//
// !INTERFACE:
//
// !USES:
//
// !INPUT PARAMETERS:
//
// !OUTPUT PARAMETERS:
//
// !DESCRIPTION:
//
// !BUGS:
//
// !SEE ALSO:
//
// !SYSTEM ROUTINES:
//
// !FILES USED:
//
// !REVISION HISTORY:
//
//  27Jun02  Username Initial specification
//
//EOP
//------------------------------------------------------------------------
//BOC
//EOC
```

4 Customizable Features in LIS

The LIS core is designed with extensible interfaces for facilitating easy incorporation of new features into LIS. The LIS core uses advanced features of the Fortran 90 programming language, which are especially suitable for object oriented programming. The object oriented style of design adopted in LIS enables the core to provide well defined interfaces or "plug points" for enabling rapid prototyping and development of new features and applications into LIS.

The LIS core includes a number of abstractions including:

- land surface model: Interfaces for adding new land surface models.
- base forcing: Interfaces for adding new model forcing schemes.
- supplemental forcing: Interfaces for adding supplemental forcing products.
- data assimilation: Interfaces for specifying assimilation of observational data using data assimilation algorithms.

The actual implementation of a component uses these abstractions following the concept of polymorphism.

4.1 What is polymorphism?

The modules in LIS are constructed using a component-based design, with each module/component designed to abstract the behavior of a certain program segment. The interfaces are designed to emulate the concept of polymorphism from the object oriented software design world. As the definition of the word implies, polymorphism is the state of being able to assume different forms. A polymorphic method is typically defined with minimal and common functionality, and specific implementations of the methods override the polymorphic method. Figure 2 shows an example of polymorphic behaviour in the real world. A *car* class is a polymorphic module, and *sports car* and *van* are specific instances of the abstract, *car* class. The *car* class might contain a *move* method, which could be overwritten with a different behaviour in the *sports car* and *van* classes. In object oriented programming, the *move* method is always invoked on the polymorphic class (*car*), and depending on the specific instance used in the simulation, the *move* call will be delegated to the *move* call of the *sports car* or the *move* call of the *van* class.

Unfortunately, true polymorphism and the automatic delegation of the polymorphic methods to the specific instances are true object oriented features. Since Fortran 90 is not an object oriented language, polymorphism can only be simulated only in software. This is achieved by the use of virtual function tables. The virtual function tables maintain a list of specific instances of each polymorphic method. Since Fortran is not an object oriented language, the task of adding the functions or "registering" the functions into the

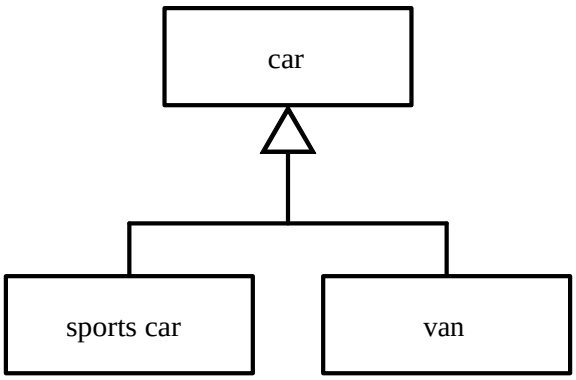

Figure 2: Example of polymorphic behaviour

virtual function tables need to be performed to simulate polymorphism. The C language allows the capability to store functions, table them, and pass them as arguments. The Fortran 90 programming language allows passing of functions as arguments. By combining these features of both languages, LIS uses a complete set of operations with function pointers.

4.2 Polymorphism in LIS

Polymorphism is simulated in LIS using virtual function tables and the actual delegation of the calls are done at run-time by resolving the function names from the table. Figure 3 illustrates how the function tables work. A function is stored in the table typically by a `register` function, that simply stores the pointer to the function at the specified index. The *call register(1,f1)* stores the function *f1* into the function table against index of 1 and the call *call register(2,f2)* stores the function *f2* into the function table with against an index of 2. When the function needs to be accessed, a generic call is made which resolves into a specific call depending on the index specified. In this case, the *call retrieve(1)* invokes the method *f1* from the table and the call *call retrieve(2)* invokes the method *f2*. This implementation helps in defining generic calls in programs. In the following, "registry" is used to refer to a function table.

In the LIS architecture, the customizable features listed in section 4 are implemented using the virtual function tables. For each customizable feature, abstract interfaces are provided by LIS, with the specific implementations and the addition to the corresponding registry left to the user. Once the functions are implemented and added to the registry, the appropriate delegation and linkages of the calls are handled by the LIS core software. Further, when a new feature is implemented in LIS, the user does not necessarily have to be familiar with the implementation details of the rest of the software. This feature enables rapid prototyping and testing of new applications into LIS.

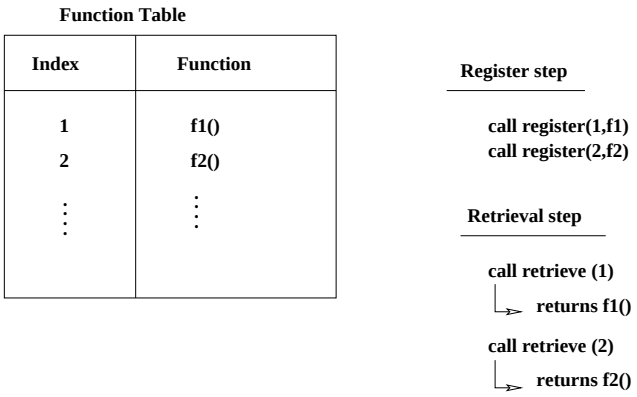

Figure 3: Example of a function table implementation

Directory Name	Synopsis
core	LIS driver routines
baseforcing	Routines to call model forcing methods
suppforcing	Routines to call supplemental forcing products
lsms	Contains land surface model codes
dataassim	Contains data assimilation methods
domains	Contains routines for creating tile-based running domains
interp	Contains spatial interpolation/reprojection methods
params	Contains routines for reading parameter data
plugins	Routines that define registries for extensible features

Table 1: Top level directory structure of LIS source code

The LIS 4.1 source code available from the LIS website contains a number of sub directories, which are organized as components. The top level organization of the source (*src*) are listed in Table 1. The *plugins* directory contains modules where the registries for each polymorphic method are defined.

5 Generic data structures in LIS

In LIS, the land surface model executions are defined on a fundamental unit called 'tile'. Each of these tiles can be mapped to a grid point on the modeling domain. LIS also supports nesting or concurrent execution on multiple domains. Each of these nests consists of a number of model tiles. The following are the key LIS variables that are related to the model execution space:

`lis%nch(n)`	Number of tiles, for each processor, for the nest `n`
`lis%glbnch(n)`	Number of tiles for the whole domain, for the nest `n`
`lis%ngrid(n)`	Number of grid points, for each processor, for the nest `n`
`lis%glbngrid(n)`	Number of grid points for the whole domain, for the nest `n`
`lis%lnc(n)`	Number of columns in the domain, for each processor, for the nest `n`
`lis%lnr(n)`	Number of rows in the domain, for each processor, for the nest `n`
`lis%gnc(n)`	Number of columns in the domain, for the whole domain, for the nest `n`
`lis%gnr(n)`	Number of rows in the domain, for the whole domain, for the nest `n`

Note that when a single processor is used, the `lis%nch(n)`, `lis%ngrid(n)`, `lis%lnc(n)` and `lis%lnr(n)` will be exactly equal to `lis%glbnch(n)`, `lis%glbngrid(n)`, `lis%gnc(n)` and `lis%gnr(n)`, respectively. Further, when no subgrid-tiling is used, the tile space and grid space are exactly equivalent (`lis%nch(n)` equals `lis%ngrid(n)`). These indices also represent the grid and tile space domain decomposition.

`lisdom(n)%gindex` defines the mapping between the grid location and the tile index.

```
tileindex = lisdom(n)%gindex(col,row)
```

where `col` and `row` are the column and row index of the grid point and `tileindex` defines the index of the tile.

The *src/core* directory contains a number of modules that provides variables that may be required while defining land surface model specific routines. Some of the useful modules and the variables provided by them are listed below. For more details, please refer to the source code documentation.

Module name	Provides
listime_manager	Variables and routines for time management
lisdrv_module	`lisdom`: representations of lis domain including tiles, grids and 2D mappings `config_lis`: overall runtime configuration `lis`: representation of overall simulation control
spmdMod	Variables and routines that define domain decomposition
constantsMod	specification of global constants
grib_module	grib support implementations
drv_output_mod	generic routines for parallel I/O

6 How to add a new land surface model in LIS?

The *plugins* directory contains the *lsm_pluginMod* module that can be used to customize and define land surface models in LIS. The *lsm_pluginMod* contains a *lsm_plugin* method that defines a number or registries to capture the basic offline operations of a land surface model. The registries can be used to define functions to perform the following tasks:

- initialization:
  Definition of land surface model variables, allocation of memory, reading run-time parameters, etc.
- setup:
  Initialization of land surface model parameters.
- dynamic setup:
  Routine to initialize or update time dependent parameters.
- run:
  Routine to execute land surface model for a single timestep.
- write restart:
  Routine to write restart files
- read restart:
  Routine to read restart files
- output:
  Routine to write output
- transfer of forcing data to model tiles:
  Routine that provides an array of forcing variables for each gridcell.
- Finalize:
  Routine that cleanups any allocated memory structures

A new LSM (lets say Noah) should implement each of the above 8 methods for successful incorporation in LIS.

The following example shows how the registry functions are defined for the Noah LSM.

```
call registerlsmini(noahId,noah_varder_ini)
call registerlsmsetup(noahId,noah_setup)
```

```
call registerlsmdynsetup(noahId,noah_dynsetup)
call registerlsmrun(noahId,noah_main)
call registerlsmrestart(noahId,noahrst)
call registerlsmoutput(noahId,noah_output)
call registerlsmf2t(noahId,retroId,noah_f2t)
call registerlsmwrst(noahId,noah_writerst)
call registerlsmfinalize(noahId, noah_finalize)
```

The `noahId` refers to the integer index assigned to Noah. The file *pluginIndices.F90* defines the conventions used in LIS. Please note that these can be modified by the user if a different convention is to be followed.

The registry functions defined for noah are:

`noah_varder_ini`	Initialization for Noah
`noah_setup`	Sets up Noah's parameters
`noah_dynsetup`	Sets up Noah's time dependant parameters
`noah_main`	Runs the Noah model on the model tiles for a single timestep
`noahrst`	Reads the Noah restart files
`noah_output`	Writes output of Noah runs
`noah_writerestart`	Writes Noah's restart files
`noah_f2t`	Transfers forcing data to Noah model tiles
`noah_finalize`	Cleanups up allocated memory structures

The first step is to organize the LSM code so that the actual model physics can be isolated to the execution on a single model tile, for a single timestep. The subroutine `SFLX` in Noah represents such a routine. The call to the model physics should be defined in `noah_main` as follows:

```
do t=1,lis%nch(n)
   call SFLX ( <arguments> )
enddo
```

Since the model prognostic variables, input parameters, and diagnostic outputs need to be accessed for initialization, output, model restart and other functions, they are defined as module variables in `noah_module`, which represents the variable definition for a single model tile. In the initialization routine (`noah_varder_ini` the memory structures are allocated as shown below (similar to the LIS structure, memory for the nests are allocated and then memory for model tiles are allocated). Finally the `readnoahcrd` call reads the run-time specifications that are specific to Noah LSM. These config options specifies variables such as locations of land surface model specific parameter files, output writing intervals, initial conditions, etc. The routine to read these variables is typically done during initialization of the land surface model.

```
allocate(noah_struc(lis%nnest))
call readnoahcrd()
do n=1,lis%nnest
   allocate(noah_struc(n)%noah(lis%nch(n)))
enddo
```

The `noah_setup` routine is used to define the land surface parameters used in Noah. These include parameters related to vegetation and soils. `noah_dynsetup` performs a similar function, to setup parameters that are time-dependent. These include the use of a monthly greenness and quarterly albedo climatologies.

`noahrst` and `noah_writerst` are restart reading and writing routines for Noah, respectively. These subroutines read and write the list of prognostic variables for Noah to a file, so that the model can be restarted from such a file. The variables are written out in tilespace, using the generic routines specified in *drv_output_mod* file.

The `noah_output` routine writes a diagnostic output file (in binary/grib/netcdf) format. The variables written out conform to the Assistance for Land Modeling Activities (ALMA; [3])) standard.

`noah_f2t` is a routine that translates the input forcing to the actual Noah model tiles. The forcing variables processed by LIS are in tile space so that the translation is a 1-to-1 mapping.

Finally the `noah_finalize` is a cleanup routine that cleanly deallocates the memory structures allocated specific to Noah.

This set of routines completes the incorporation of Noah LSM in LIS. A number of LSMs are implemented using this plugin style.

7 How to add a new forcing scheme in LIS?

The boundary conditions describing the (upper) atmospheric fluxes are known as "forcings". LIS makes use of model derived data as well as satellite and ground-based observational data as forcings. The land surface models are typically run using model derived data. The observational data is used to overwrite the model derived data, whenever they are available. In LIS, a scheme that includes a complete set of variables defined in the ALMA forcing data convention can be used as a "baseforcing". A scheme/product that includes a subset of the ALMA forcing convention should be implemented as a supplemental forcing, if it is to be used in LIS.

The *plugins* directory contains modules *baseforcing_pluginMod* and *suppforcing_pluginMod*, that can be used to customize and define base forcing schemes and supplemental forcing schemes, respectively. These modules provide the plugin routines *baseforcing_plugin*, and *suppforcing_plugin*, respectively.

baseforcing_module and *suppforcing_module* provides registries to define functions to perform the following tasks.

- definition of native domain:
  Routines to define the native domain of the forcing data, read run-time specific parameters through a namelist, etc.
- retrieval of forcing data:
  Routines to retrieve the forcing data, and interpolate them.
- temporal interpolation:
  Routines to interpolate data temporally.
- finalize:
  Routines to cleanup

The following code segment shows how two baseforcing schemes are included in LIS.

```
call registerdefinenative(gdasId,defineNativeGDAS)
call registerget(gdasId,getgdas)
call registertimeinterp(gdasId,time_interp_gdas)
call registerforcingfinal(gdasId,gdasforcing_finalize)
```

Similar to the case in *lsm_pluginMod*, the indices used in the registries need to be the same for a particular scheme. The `gdasId` is defined in the file *pluginIndices.F90*.

The input forcing has to be spatially and temporally interpolated to the LIS grid and the timestep used for LSM simulations. For computational performance considerations, spatial interpolation is an expensive operation, primarily because of the computation of interpolation weights. As a result, spatial interpolation is broken up into two steps: (1) computation of interpolation weights and (2) actual spatial interpolation.

The `defineNativeGDAS` routine performs two main functions. (1) allocate the required memory structures and read runtime configurable options, and (2) to setup the interpolation weights required for interpolating the input forcing to the LIS grid.

The `getgdas` reads the input forcing data based on the model clock time, and interpolates it to the LIS model grid, using the interpolation weights defined in the `defineNativeGDAS`. A number of generic interpolation algorithms (bilinear, conservative, and neighbor) are provided in the *src/core* directory.

The `time_interp_gdas` routine temporally disaggregates the spatially interpolated forcing data to the model timestep. This step includes the interpolation of forcing data between two or three forcing data intervals. The disaggregation is typically a weighted average. For downward shortwave radiation, a zenith angle-based disaggregation is typically performed.

Finally the `gdasforcing_finalize` specifies the cleanup or deallocation of allocated memory structures specific to GDAS forcing.

8 Customizing LIS for data assimilation

The emphasis of land surface data assimilation is to ingest remotely-sensed observations such as temperature, soil moisture, and snow to adjust the model representation which is most consistent with the observations. The data assimilation plugins in LIS are designed to support the use and implementation of different sequential algorithms in land surface model simulations.

The generic data assimilation plugin in LIS is enabled by the combination of a number of abstractions. There are many different data assimilation algorithms (direct insertion (DI), extended kalman filter (EKF), etc.) that can be employed. The chosen algorithm is used to update the relevant state variable(s) of the land surface model being employed. Finally, the observational data used in assimilation can be obtained from many different sources. Figure 4 shows the interactions of these three abstractions. LIS core enables the integrated use of these abstractions through explicitly defined interfaces. All data exchanges between the data assimilation components are enabled using the constructs provided by the Earth System Modeling Framework (ESMF; [4]). ESMF provides a standardized, self-describing format for data exchange between model components through the $ESMF\_State$ datatype. The three abstractions shown in Figure 4 exchange information with each other using $ESMF\_State$ objects.

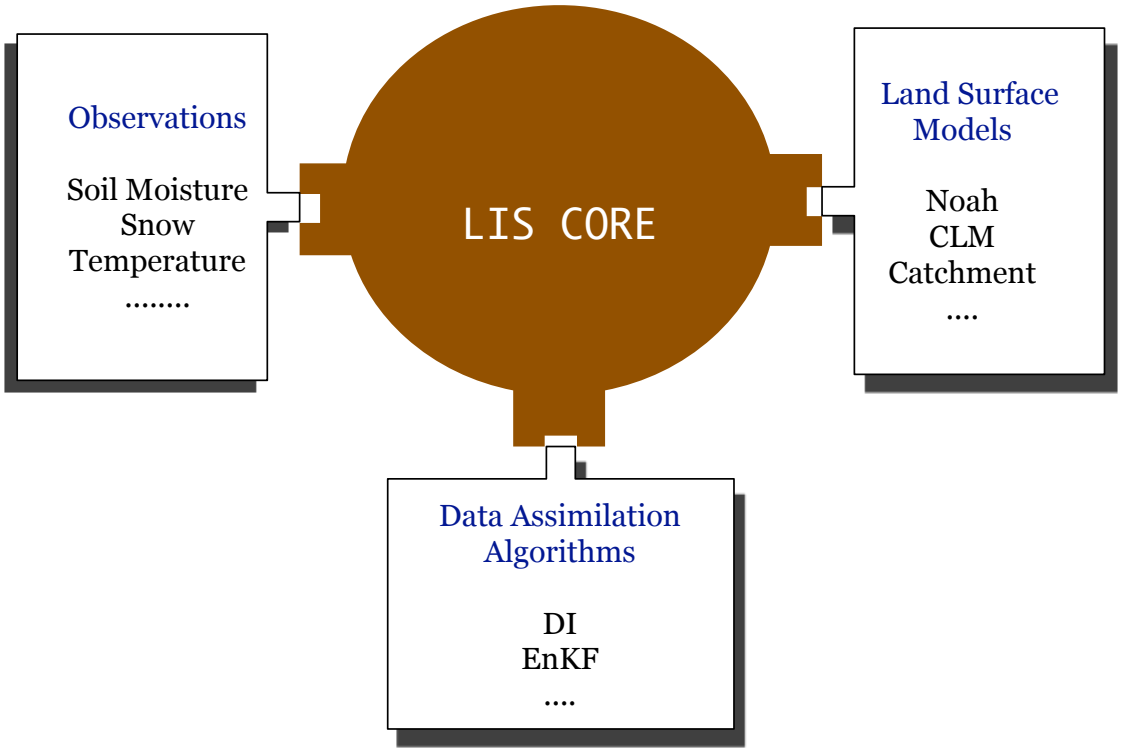

Figure 4: Data assimilation abstractions and their interactions in LIS

The sequential data assimilation techniques typically involve updating the estimate of the system state at each observation time, based on the measurements up to this time. The overall process can be represented using a number of equations, as described in this section. Using the notation used in [8], the nonlinear land surface is represented in the generic form

$$\mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k) + \mathbf{w}_k \tag{1}$$

where $\mathbf{x}_k$ represents the state vector at time k, $\mathbf{f}(.)$ is the nonlinear operator, and $\mathbf{w}_k$ represents the uncertainties due to errors in the model formulation and boundary conditions. The observations at time k denoted by $\mathbf{y}_k$ is connected to the system states by the equation

$$\mathbf{y}_k = \mathbf{H}_k(\mathbf{x}_k) + \mathbf{v}_k \tag{2}$$

where the operator $\mathbf{H}_k$ translates the system states to the measurement variables. Measurement errors are represented in the term $\mathbf{v}_k$. The noises in $\mathbf{w}_k$ and $\mathbf{v}_k$ are typically assumed to be independent random vectors with mean zero and covariances $\mathbf{Q}_k$ and $\mathbf{R}_k$, respectively. The difference between the predicted observation vector and the measurement vector $(\mathbf{y}_k - \mathbf{H}_k(\mathbf{x}_k))$ known as the 'innovations vector' is used to make a correction to the system states to generate an improved state estimate $\mathbf{x}_{k+1}$, known as the analysis, represented by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{K}(\mathbf{y}_k - \mathbf{H}_k(\mathbf{x}_k) + \mathbf{v}_k) \tag{3}$$

where $\mathbf{K}$ represents the "gain matrix", which is chosen to ensure that analysis states converge to the true states of the system over time. The model is then evolved forward again from the analysis states to the next time where an observation is available and the process is repeated.

In order to define a custom data assimilation instance, a number of routines need to be specified in three different registries: (1) `dataassim_plugin` in *dataassim_pluginMod* that specifies the algorithm for data assimilation, (2) `dataobs_plugin` in *dataobs_pluginMod* that specifies the observation for data assimilation and (3) `lsmda_plugin` in *lsmda_pluginMod* that specifies the LSM related interfaces for data assimilation.

dataassim_plugin defines the following registries:

- Init:

  Defines routines for initializing memory structures and other initializations.

- Assimilate/Update:

  Method that provides the assimilation/update algorithm.

- Output:

  Method to write data assimilation diagnostics to a file

- Finalize:

  Method to cleanup allocated memory structures

These methods are registered using a single index for the assimilation algorithm that corresponds to the implementation.

The data assimilation algorithm implementations interact with the land surface models during the process of modifying and updating the state variables. The interaction is dependent on the land surface model and the state variable being updated. To facilitate this interaction, LIS provides a number of plugin interfaces in the *lsmda_pluginMod*. These interfaces are abstractions of the querying and updating operations needed to enable interaction with land surface models. Each land surface model used for data assimilation needs to extend these querying and updating interfaces, in addition to the interfaces pertaining to the basic operation of a land surface model described in section 6. A list of the required interfaces for each LSM are listed below.

- get state variables method to package the list of prognostic variables into an ESMF State.
- set state variables method to translate the given ESMF state and update the prognostic state variables.
- QC LSM state method to QC a given ESMF state (check bounds, physical consistency, etc)
- define "obspred" method to that defines the "obspred", which is the model's prediction of what the observations should be.
- Scale LSM state method to scale the LSM variables, if needed, so that the matrices used in an algorithm such an EnKF are well formed.
- descale LSM state opposite of the scaling method, to convert variables back to the original space.
- update LSM state This routine specifies how the analysis increments are to be applies. The routine provides flexibility in applying the increments to the selective list of variables.

These methods are registered using two indices. 1. index for the land surface model and 2. "Assimilation set", which refers to a combination of observation and the variables being updated. For example, AMSR-E soil moisture observation to update Noah LSM variables, AMSR-E soil moisture observations to update Catchment LSM variables and MODIS snowcover observations to update Noah LSM variables constitute different assimilation sets.

Another generalization associated with the data assimilation operations is related to the handling of the observational data. The generic plugins implemented in LIS for this are designed similar to the handling of other (parameter, forcing) datasets.

The method *dataobs_plugin* defined in *dataobs_pluginMod* defines the registries for the functions related to observation handling. *dataobs_plugin* defines the following registries:

- Setup:

  Defines routines for initializing memory structures and other initializations.

- Reading method:

  Method that reads the observation data and packages it into an ESMF state

- Get number of selected observations:

  Returns the number of selected observations to be used for a single grid point.

These methods are registered using a single index for the assimilation set that corresponds to the method.

9 "Use only what you need"

A key advantage of the use of function tables for simulating polymorphism is the ability to use only the components that is needed. The "plug and play" of different components allow LIS to remain flexible, rather than evolve into a monolithic software when new components and features are added. This section describes how to compile and use only the needed components.

9.1 Defining source directories for compilation

A file called *Filepath* in the *$WORKING/LIS/src/make* directory specifies all the source files that will be included during compilation. A sample *Filepath* is shown below.

```
../core
../domains
../interp
../params/soils
../params/landcover
../params/elev
../params/lai
../baseforcing/geos
../baseforcing/gdas
../suppforcing/cmap
../suppforcing/agrmet
../plugins
../lsms/noah.2.6
../lsms/mosaic
../lsms/vic
../lsms/clm2
../lsms/clm2/main
../lsms/clm2/biogeophys
../lsms/clm2/biogeochem
../lsms/clm2/camclm_share
../lsms/clm2/csm_share
../lsms/clm2/riverroute
../lsms/clm2/ecosysdyn
```

9.2 Defining components while building the executable

As described in the previous sections, the specific instances of each customizable interface in LIS are defined in different registries. Once the user specifies the components to be used in these interfaces, the *Filepath* directory can be modified to include only these components. For example, if a user is interested in running only one land surface model (say Noah) and does not want to keep and compile other land surface models, the *Filepath* can be modified to indicate that only Noah needs to be compiled as follows (Note that CLM, Mosaic, and vic directories are excluded):

```
../core
../domain-plugin
../domains
../interp
../plugins
../params/soils
../params/landcover
../params/elev
../params/lai
../baseforcing/geos
../baseforcing/gdas
../suppforcing/cmap
../suppforcing/agrmet
../lsm-plugin
../lsms/noah.2.6
```

Correspondingly, the *lsm_plugin* method in *$WORKING/LIS/src/lsm-plugin/lsm_pluginMod.F90* needs to be defined as (excluding other land models from the registry):

```
subroutine lsm_plugin
  use noah_varder, only : noah_varder_ini
  external noah_main
  external noah_setup
  external noahrst
  external noah_output
  external noah_f2t
  external noah_writerst
  external noah_dynsetup
```

```
    call registerlsmini(1,noah_varder_ini)
    call registerlsmsetup(1,noah_setup)
    call registerlsmdynsetup(1,noah_dynsetup)
    call registerlsmrun(1,noah_main)
    call registerlsmrestart(1,noahrst)
    call registerlsmoutput(1,noah_output)
    call registerlsmf2t(1,noah_f2t)
    call registerlsmwrst(1,noah_writerst)
  end subroutine lsm_plugin
```

Similarly, different combinations of using the components can be implemented defining the registries appropriately and specifying the corresponding source files in the *Filepath* file.

References

[1] Community climate system model, software developers guide. http://www.ccsm.ucar.edu/csm/working_groups/Software/

[2] Protex documentation system. http://gmao.gsfc.nasa.gov/software/protex.

[3] ALMA. http://www.lmd.jussieu.fr/ALMA/.

[4] ESMF. http://esmf.ucar.edu.

[5] S.V. Kumar, C.D. Peters-Lidard, J.L. Eastman, and W.-K. Tao. An integrated high resolution hydrometeorological modeling testbed using lis and wrf. *Environmental Modeling and Software*, 23:169–181, 2007.

[6] S.V. Kumar, C.D. Peters-Lidard, Y. Tian, P.R. Houser, J. Geiger, S. Olden, L. Lighty, B. Doty, P. Dirmeyer, J. Adams, K. Mitchell, E.F. Wood, and J. Sheffield. Land information system - an interoperable framework for high resolution land surface modeling. *Environmental Modeling Software*, 3(3):157–165, 2006.

[7] S.V. Kumar, R.H. Reichle, C.D. Peters-Lidard, R.D. Koster, X. Zhan, W.T. Crow, J.B. Eylander, and P.R. Houser. A land surface data assimilation framework using that land information system: Description and applicatins. *Advances in Water Resources*, page in print, 2008.

[8] R. H. Reichle, J. P. Walker, R. D. Koster, and P. R. Houser. Extended versus ensemble kalman filtering for land data assimilation. *Journal of Hydrometeorology*, 3(6):728–740, 2002.